2013W1-lecture21

November 1, 2013

Contents

1 Question of the Day

TypeScript is a superset of JavaScript that provides static type annotations—information about the types of identifiers provided statically.

Why bother with a static type system?

1.1 Solution :solution:

The usual answer given is to find errors early. I hope you have experienced (as I have!) accidentally forgetting to call `desugar` on some subexpression inside your desugarer and having the static type system warn you that the resulting program does not pass type-checking. It can tell because leaving out the call to `desugar` means using an `ExprS` where an `ExprC` is expected.

This sort of information can be quite useful.. but it can also be quite annoying! Consider that in Racket, for instance, it's easy to leave part of a function unimplemented. You can just fill in `...` or `false` or just about anything you like in that part. It won't fail type-checking, and it won't cause a problem unless that incorrect code happens to be run. That's great for *prototyping*: quick, early drafting of design ideas and functionality.

However, that's **not** what TypeScript is for. In fact, TypeScript allows the programmer to do many things that most statically-typed languages would object to.

Instead, the type annotations in TypeScript are meant to provide information to the programmer: to improve their interface. With the type annotations, the development environment is able to provide context-sensitive help and suggestions like completing function names and providing the function's argument list with types.

2 Logistics

2.1 Conceptual Assignment #4

Corrections due Tue at 8PM.

As always, even if you got everything correct, HAND IN to let us know that!

2.2 NO QUIZ on Wed

2.3 Programming Assignment #4

Full version due Sunday.

No demo, sadly. (Between the midterm and the Nov 11 holiday, there's just no good week to do it!)

Instead, shortly after the midterm, we'll let you know the three core syntax variants we want you to report on and what we want to know, and you'll handin what you would have said at a demo. We'll likely also ask an open-ended question.

2.4 Midterm prep materials coming soon

We'll definitely have some pre-posted background (like last time). We also plan to provide some reasonably well-posed questions you can work through, although we don't promise they'll be as tightly scoped as exam questions.

3 Boxing Day

We've implemented boxes in our interpreter, which means introducing a store.

But, what does it really mean to "play with" store locations? Once we have mutation and a distinction between the environment and the store, the semantics of a language can become much more confusing. We should understand how for at least two reasons:

1. We're all about semantics!
2. It will help us understand why we should **limit our use of mutation and state** when writing programs in any language.

(Note: I'm not saying "don't use mutation". I'm saying to understand why to limit your use of it!)

3.1 A Box By Any Other Name

What would the following Java method print?

```
public void play() {
    int x;
    x = 1;
    int y = x;
    y = y * 10;
    System.out.println(x + y);
}
```

Now, what does the following Java program print?

```
class Box {
  public int value;
}
```

```
public class Play {
  public static void main(String[] args) {
      Box x = new Box();
      x.value = 1;
      Box y;
      y = x;
      y.value *= 10;
      System.out.println(x.value + y.value);
  }
}
```

How about this C++ program:

```
class Box {
public:
  int value;
};

int main(int argc, char * argv[]) {
  Box x;
  x.value = 1;
  Box y = x;
  y.value *= 10;
  std::cout << (x.value + y.value) << std::endl;
}
```

3.2 Brief overview of C++ store location syntax/semantics

C++ is handy because it really lets us explore the store. It gives us `*` and `&`—which are both type modifiers and, completly separately, operators, one of them two entirely different operators!—-which allow us to specify very precisely where we want something to be a value and where we want it to be a store location. We can do this because store locations are also legitimate values in C++.

Let's review part of what C++ offers. Remember that types are sets of values.

First, as in almost any language with mutable variables, for the expression on the left side of an assignment operator, the rules of evaluation change slightly. Instead of looking up the value at the store location indicated by the expression, we just get the store location and update that location to contain a copy of the value from the right side of the assignment.

And:

- `T x;` where `T` is a type and `x` is an identifier: declares a new variable named `x` of type `T`. `x` has a store location and the entire `T` value is stored at that store location, regardless of the type of `T`.
  - This is like Java for lowercase (primitive) types like `int` and `double`.
  - This is **UNLIKE** Java for uppercase (class) types like `Object` and `Box`. The C++ equivalent of Java's `Box x;` is `Box * x;`.
- `T * x;` where `T` is a type and `x` is an identifier: declares a new variable named `x` of type `T*`. `T*` is the set of store locations (addresses) of values of type `T`.
- `*e` where `e` is an expression of type `S*`, where `S` is any type: evaluates `e` to a store location and evaluates to the value at that location (interpreted as an `S`).
- `e.name` where `e` is an expression of type `R` and `name` is a field name in the type `R`: evaluates `e` to a value and then evaluates to the value of its `name` field.
  - This is **UNLIKE** Java's syntax `e.name`, which instead corresponds to C++'s `e->name`.
- `e->name` where `e` is an expression of type `R*` and `name` is a field name in the type `R`: is syntactic sugar for `(*e).name`. In other words, evaluates `e` to a store location, gets the `R` value at that store location, and evaluates to the `name` field of that value.

These semantics will let us play around very freely with store locations and values. Java, on the other hand, uses its syntax to both obscure its use and restrict our use of store locations. (This is not necessarily a bad thing!)

3.3 More Boxes By Any Other Name

So, how about this C++ program:

```
class Box {
public:
  int value;
};
```

```
int main(int argc, char * argv[]) {
  Box * x = new Box;
  x->value = 1;
  Box * y = x;
  y->value *= 10;
  std::cout << (x->value + y->value) << std::endl;
}
```

And this C++ program?

```
class Box {
public:
  int value;
};

int main(int argc, char * argv[]) {
  Box * x = new Box;
  x->value = 1;
  Box * y = x;

  Box * temp = new Box;
  temp->value = 10 * x->value;
  y = temp;

  std::cout << (x->value + y->value) << std::endl;
}
```

Next, let's try this C++ program:

```
class Box {
public:
  int * value;
};

int main(int argc, char * argv[]) {
  Box x;
  x.value = new int;
  *(x.value) = 1;
  Box y = x;
```

```
  *(y.value) *= 10;

  std::cout << (*(x.value) + *(y.value)) << std::endl;
}
```

Finally, let's try this C++ program:

```
class Box {
public:
  int * value;
};

int main(int argc, char * argv[]) {
  Box x;
  x.value = new int;
  *(x.value) = 1;
  Box y;
  y.value = x.value;
  *(y.value) *= 10;

  std::cout << (*(x.value) + *(y.value)) << std::endl;
}
```

3.4 Probing a New Semantics

Here is a `plai-typed` program:

```
(display (local [(define x 1)
                 (define y x)]
           (begin
             (set! x (* x 10))
             (+ x y))))
(display "\n")

(display (local [(define x (box 1))
                 (define y x)]
           (begin
             (set-box! y (* 10 (unbox y)))
             (+ (unbox x) (unbox y)))))
(display "\n")
```

```
(display (local [(define x (box 1))
                 (define y (box 0))]
           (begin
             (set-box! y (unbox x))
             (set-box! y (* 10 (unbox y)))
             (+ (unbox x) (unbox y)))))
(display "\n")
```

We can play the "what does this print" game, but even more important than that is "if we know what this prints, what does this tell us about the semantics of the language"?

So, this prints:

```
11
20
11
```

What does this tell us about the semantics of the language?

3.5 Passing Arguments "By Value" (the most common semantics)

By default, each of Java, C++, and `plai-typed` passes arguments "by value". That means when we apply a function, we:

1. evaluate the actual parameter expression,
2. allocate a store location for (this instance of) the formal parameter,
3. and **copy** the actual parameter value into the new store location.

(Unlike the others, C++ allows us to specify a rather different semantics, which we'll talk about next time. However, we can already understand that semantics from what we've done today!)

Let's change two each of our Java, C++, and `plai-typed` examples to use functions. In each case, bearing the "pass-by-value" semantics in mind, what does the program print and why?

(**WARNING**: of all of these, Java is the trickiest. That's because Java is underhanded and sly about where it uses store locations.)

A Java program that passes an `int` to a function that changes its formal parameter:

```
public class Play {
    public static void times10(int i) {
        i *= 10;
    }

    public static void main(String[] args) {
        int x;
        x = 1;
        times10(x);
        System.out.println(x);
    }
}
```

A Java program that passes a "boxed" `int` to a function that changes a field of its formal parameter.

```
class Box {
  public int value;
}

public class Play {
    public static void times10(Box b) {
        b.value *= 10;
    }

    public static void main(String[] args) {
        Box x;
        x.value = 1;
        times10(x);
        System.out.println(x.value);
    }
}
```

A C++ program that passes an `int` to a function that changes its formal parameter:

```
void times10(int i) {
  i *= 10;
}
```

```
int main(int argc, char * argv[]) {
  int x;
  x = 1;
  times10(x);
  std::cout << x << std::endl;
}
```

A C++ program that passes the store location of an `int` to a function that changes the value at that store location:

```
void times10(int * i) {
  (*i) *= 10;
}

int main(int argc, char * argv[]) {
  int * x = new int;
  *x = 1;
  times10(x);
  std::count << *x << std::endl;
}
```

Two for the price of one: a `plai-typed` snippet that passes a number to a function that changes its formal parameter; then, a `plai-typed` snippet that passes a boxed number to a function that changes the boxed value in its formal parameter.

```
(display (local [(define (times10 i)
                   (set! i (* i 10)))
                 (define x 1)]
           (begin
             (times10 x)
             x)))
(display "\n")

(display (local [(define (times10 b)
                   (set-box! b (* (unbox b) 10)))
                 (define x (box 1))]
           (begin
             (times10 x)
             (unbox x))))
(display "\n")
```

4 What have we learned today?

- From QotD:
  - Give some compelling examples of static analyses besides parsing/desugaring.
- Store locations
  - Given a semantics for a language that includes mutation and a short program in that language, sketch the contents of values and the store in order to illustrate the behaviour of the program.
  - Given a short program in a language that includes mutation, speculate intelligently on the semantics of the language (particularly with respect to its handling of store locations).
- Pass-by-Value
  - Explain what pass-by-value semantics means in a language.
  - Explain with examples (including solving such examples) why pass-by-value semantics does not mean "the callee function cannot make changes using its argument that affect calling function's interpretation of its actual parameter".
    - In other words: While it **is** technically true that the callee cannot change the value of the caller's actual parameter expression, explain why that technicality is not enough to guard against changes in the callee having a meaningful impact on the caller.
- More: Unnecessary State Change as a "Bad Thing"
  - Explain from a software engineering standpoint why (unnecessary or excessive) use of mutation can make software more difficult to reason about and change.

5 Some Practice Exercises

Will post midterm practice exercises very soon!

Meanwhile, try this:

- Write at least two versions of a `swap` program in `plai-typed` and Java. Each one should be intended to take two arguments and swap the values in those arguments (i.e., `x`'s new value is `y`'s old value and `y`'s new value is `x`'s old value). Only one should work, however. If you know any other programming language, do the same in that other programming language.